Zombie Memory: Extending Memory Lifetime by Reviving Dead Blocks

Rodolfo Azevedo† John D. Davis‡ Karin Strauss‡
Parikshit Gopalan‡ Mark Manasse‡ Sergey Yekhanin‡

†University of Campinas ‡Microsoft Research

ABSTRACT

Zombie is an endurance management framework that enables a variety of error correction mechanisms to extend the lifetimes of memories that suffer from bit failures caused by wearout, such as phase-change memory (PCM). Zombie supports both single-level cell (SLC) and multi-level cell (MLC) variants. It extends the lifetime of blocks in working memory pages (primary blocks) by pairing them with spare blocks, i.e., working blocks in pages that have been disabled due to exhaustion of a single block's error correction resources, which would be 'dead' otherwise. Spare blocks adaptively provide error correction resources to primary blocks as failures accumulate over time. This reduces the waste caused by early block failures, making working blocks in discarded pages a useful resource. Even though we use PCM as the target technology, Zombie applies to any memory technology that suffers stuck-at cell failures.

This paper describes the Zombie framework, a combination of two new error correction mechanisms (ZombieXOR for SLC and ZombieMLC for MLC) and the extension of two previously proposed SLC mechanisms (ZombieECP and ZombieERC). The result is a 58% to 92% improvement in endurance for Zombie SLC memory and an even more impressive 11× to 17× improvement for ZombieMLC, both with performance overheads of only 0.1% when memories using prior error correction mechanisms reach end of life.

Categories and Subject Descriptors

B. Hardware [**B.3. Memory Structures**]: B.3.4. Reliability, Testing and Fault-Tolerance; E. Data [**E.4. Coding and Information Theory**]: Error control codes

General Terms

Reliability

Keywords

Error Correction, Drift Tolerance, Phase-Change Memory

ISCA'13 Tel-Aviv, Israel

1. INTRODUCTION

The current technology roadmap shows that scaling DRAM to smaller features [16] is rapidly slowing down. Fortunately, DRAM replacement solutions are emerging [29, 28, 33]. These solutions provide more stable storage and are based on magnetic or physical properties of materials. Phase-change memory (PCM) [12, 1, 3], a resistive memory technology, is already shipping as a NOR-Flash replacement. It is faster, uses less power, and achieves longer lifetimes than Flash.

Using PCM as a DRAM replacement for main memory, however, poses challenges. One such challenge is the endurance of individual bits. While DRAM cells typically support an average of 10^{15} writes over their lifetime, PCM cells last for as little as 10^8 writes on average. Permanent DRAM cell failures are so rare that mechanisms to tolerate them are wasteful: They disable the entire physical page where the failure occurred, which then becomes unavailable for software use. Since PCM cells wear out much faster, using the same approach would quickly disable all PCM pages. Thus, to make PCM a viable alternative for main memory, lifetime-extending mechanisms are crucial for both single-level cell (SLC) and multi-level cell (MLC) PCMs. An additional challenge with MLC PCM is drift: Once written, cell resistance may change over time. This adds complexity to MLC error correction mechanisms because they must tolerate both wearout and drift.

Multiple hardware-only error correction mechanisms tackle wearout by transparently correcting and hiding failures from software layers [15, 30, 31, 37, 24]. Despite significant progress, these mechanisms remain inefficient, wasting a large number of working bits when they can no longer correct errors and thus must disable pages.

This paper proposes the Zombie framework, which lets a variety of error correction mechanisms use the abundant working bits in disabled pages to extend the lifetime of pages still in service. Its unifying principle is to pair a block, or even a subblock, sourced from disabled pages (*spare blocks*) with a block in software-visible pages (*primary blocks*), extending the primary block's useful lifetime (and turning them into 'zombies'). Zombie enables on-demand pairing and gradual spare subblock growth, i.e., primary blocks are paired with spare blocks only when they exhaust their own error correction resources and can gradually increase their spare subblock size as additional resources are needed. Although this paper focuses on PCM memory, Zombie applies to other memory technologies that suffer from stuck-at cell failures.

This paper also proposes two new error correction mech-

anisms, ZombieMLC and ZombieXOR, and extends two existing ones (ZombieECP and ZombieERC) to showcase the Zombie framework. ZombieMLC, as the name suggests, is designed specifically for MLC and, to our knowledge, is the first mechanism to tolerate both drift and stuck-at failures. The other mechanisms (ZombieXOR, ZombieECP and ZombieERC, collectively called "ZombieSLC mechanisms") tolerate only stuck-at failures and are better suited to SLC PCM.

To illustrate the opportunity Zombie leverages, Figure 1 shows the average number of bit flips (or writes to a bit) and the average fraction of failed bits when a page must be disabled, for several previously proposed mechanisms that protect SLC PCM. The fraction of failed bits measures the amount of waste — the lower this fraction, the higher the number of unusable working bits that are wasted. All practical mechanisms waste at least 99% of the bits in a page. Oracle64 and Oracle128 represent ideal mechanisms that correct 64 and 128 bit failures per block, respectively. Even though increasing the error tolerance from 64 to 128 bit failures dramatically reduces bit waste, it is arguably in the diminishing returns region with respect to increasing the number of bit flips. Note that Oracle64 increases memory endurance (68 million flips) by 50% or more compared to other previously proposed mechanisms (SAFER at 44 million flips), representing a significant memory lifetime improvement opportunity that can be realized by the Zombie framework. The opportunity is significantly larger for MLC PCM.

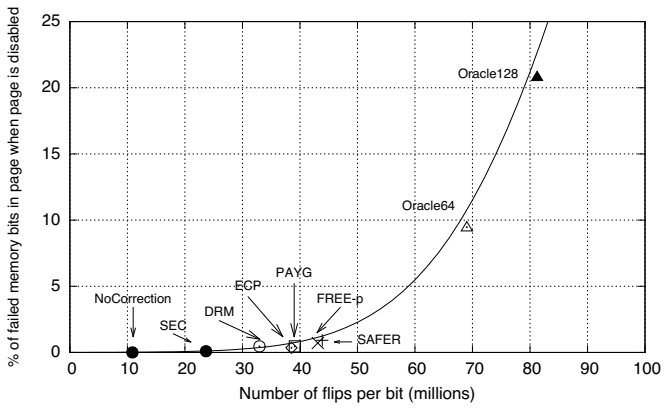


Figure 1: Average number of bit flips (x-axis) and average fraction of failed bits (y-axis) when a page must be disabled for a variety of error correction mechanisms that protect an SLC PCM (represented by markers), assuming an average cell lifetime of 10^8 bit flips (writes) and a 0.25 coefficient of variance. The solid line is the cumulative distribution function of failed memory bits as a function of bit flips.

By leveraging this opportunity, the ZombieSLC mechanisms extend memory lifetimes by 58% to 92%. ZombieMLC achieves even more impressive lifetime extensions — of 11× to 17×— while also tolerating drift. Zombie achieves these longer lifetimes with graceful degradation and at low performance overhead and complexity.

The rest of this paper is structured as follows. Section 2 presents background on PCM and the relevant previous error tolerance approaches. Section 3 provides an overview of the various Zombie mechanisms, while Section 4 offers more implementation details. Section 5 evaluates Zombie, Section 6 reviews related work and, Section 7 concludes our discussion.

2. BACKGROUND

2.1 Phase-Change Memory Basics

The storage principle in PCM leverages the phase of the material in a cell: Depending on how cells are heated up and cooled down, the material can become amorphous or crystalline, which changes its conductivity. This allows the value to be read out by running a small current through the cell and measuring the voltage. The material can also be partially crystallized to create multi-level cells that store more than one bit.

PCM differs from DRAM in multiple aspects. Besides having higher write operation latency and energy, PCM has a very different failure profile[1]. PCM cells are less vulnerable to soft errors than DRAM cells. Prior research has shown that PCM retention time will reach 10 years at 85^oC, and that the thermal interference between cells is very low even over a 10-year retention period [5]. However, multi-level cells are subject to a phenomenon called *drift*, which causes the resistance of cells to grow over time [14, 23], potentially reaching a different level and resulting in transient errors. Other challenges specific to MLC PCM, like spontaneous crystallization, can be handled at the device level [13].

Temperature cycles in write operations also make PCM much more susceptible to wear; PCM currently supports on the order of 10^8 bit flips per cell. This poses one of the most significant challenges in using PCM as main memory. Consequently, writes are performed differentially, i.e., a block is read and compared to the value to be written so that only the bits that changed are actually modified. In addition, errors resulting from wearout failures can be detected at write time and are "stuck-at" faults. A verification operation, performed after every write, reads the block again and compares the new value to the expected value. PCM write operations are thus a three-step process: read-write-verify[2].

There have been many prior error correction proposals for SLC PCM [15, 30, 31, 37, 24, 7]. Except for DRM [15], none of these proposals attempt to reuse bits in pages that have already been disabled to correct bits in pages that are still enabled. DRM reuses bits at the granularity of pages. Zombie mechanisms reuse them at the block or subblock granularity level. This finer granularity enables more flexibility in correcting errors, enhancing page longevity and making memory degradation more gradual. More detail on prior mechanisms is provided in Section 6.

2.2 Zombie-Extended Correction Mechanisms

ECP [30] is based on adding replacement cells and pointers to the failed cells they replace. The original ECP proposal adds 12.5% extra bits to each block and uses them to store 6 replacement entries and a used-entries counter, as shown in Figure 2. ZombieECP extends ECP by providing additional replacement entry storage, up to an entire spare block.

[1]This paper uses error, fault and failure interchangeably to refer to wearout, stuck-at faults.

[2]We assume differential writes, read-write-verify write operations, and perfect wear leveling.

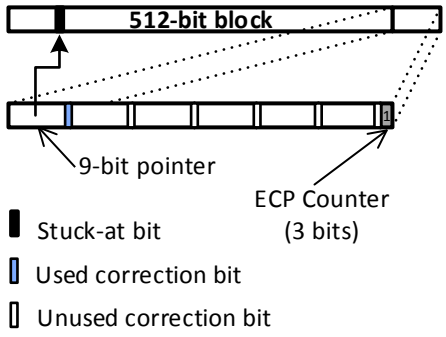


Figure 2: A 512-bit PCM block with 6 associated ECP pointers. The error correction overheard is 12.5%.

Codes to deal with stuck-at faults use one-to-many encoding functions. Multiple possible representations of the same message ensure that at least one representation is appropriately aligned with the values of stuck cells and can actually be stored in memory. This paper uses the notation $[n, k, d]$ to denote coding schemes that encode k-symbol messages to n-symbol codewords and tolerate $d-1$ stuck-at faults.

ZombieERC implements one such mechanism proposed by Tsybakov [34], optimizing it for practical use in the PCM context. To store k-bit messages in an n-bit memory block where up to $d-1$ cells are stuck, ZombieERC uses a binary $k \times n$ matrix G, which is a generator matrix of a binary linear code of length n, dimension k, and distance d [22]. The possible representations of x are all vectors y such that $Gy^T = x$. If more than one vector y is aligned with the values of stuck cells, ZombieERC prefers the vector y that optimizes memory lifetime (i.e., wears the spare block).

Rank modulation [23, 4, 18], an encoding technique that tackles the drift issue in MLC memory, constructs a codeword as a string of relative ranks of values stored in a group of cells such that their specific permutation compared to a base string (e.g., monotonically increasing ranks) encodes the original data message. Using small groups results in a low probability of the cells in that group drifting enough to change their relative rank. Although rank modulation mitigates drift, it does not tolerate stuck-at cells [23]. Naïvely combining rank modulation with error correction does not result in a mechanism capable of tolerating both drift and wearout. ZombieMLC uses certain principles from rank modulation to mitigate drift, yet it can also tolerate wearout.

3. THE ZOMBIE FRAMEWORK

Zombie is a hardware framework that allows error correction mechanisms to recycle good cells in disabled (or dead) pages to extend the life of pages that are still in service (or alive) but are running out of error correction resources. Zombie thus leverages the high number of good cells left in pages that have to be disabled due to a small number of failures that cannot be corrected, as illustrated in Figure 1.

3.1 Pairing Strategies

Initially, all memory pages are independent and visible to software. When the first block runs out of intrinsic error correction resources, its entire page is disabled and made unavailable to software. Zombie recycles good blocks in disabled pages (spare blocks or subblocks) by pairing them individually with blocks in working pages that are about to run out of error correction resources (primary blocks), as illustrated in Figure 3. The spare block increases the error correction resources of its primary block and keeps it alive longer. The various Zombie mechanisms differ in how they combine the two blocks to implement error correction, but all use this pairing strategy.

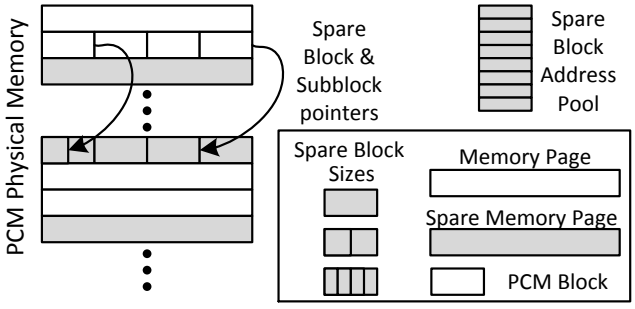


Figure 3: Primary blocks (in white) pointing to a spare subblock (left) and block (right), in gray. The spare block may be the same size as the primary block or smaller: half, a quarter or an eighth of the size. Each spare block size has it own address pool.

Many of the Zombie mechanisms are *adaptive*: Instead of initially pairing blocks with a full spare block, a primary block may be paired with a smaller spare subblock, as shown in Figure 3. This ensures better utilization of spare blocks because primary blocks get only as much extra resources as they need. As more resources are needed, primary blocks are paired with increasingly larger subblocks until they reach the size of an entire block. The memory controller keeps one pool per size and hands out spare subblocks for on-demand pairing.

3.2 Error Correction Mechanisms Overview

This paper showcases Zombie with two new error correction mechanisms (ZombieMLC and ZombieXOR) and two existing ones [30, 34] that were extended to leverage the framework (ZombieECP and ZombieERC). These mechanisms belong to one of two classes. ZombieXOR, ZombieECP and ZombieERC correct stuck-at cells but cannot tolerate drift, which is typical of MLC. Thus, they are more suitable to SLC (or MLC memories without drift or similar issues), so we refer to these mechanisms as ZombieSLC. The remaining mechanism is, to our knowledge, the first to tolerate both drift and stuck-at errors, and we refer to it as ZombieMLC. Basic descriptions of each Zombie error correction mechanism follow. Section 4 provides more detail about each.

ZombieXOR. ZombieXOR pairs two blocks cell-by-cell. Once blocks are paired, ZombieXOR relies on simple XOR-based encoding to read and write these blocks. If a given cell in a primary block is stuck, its counterpart in the spare block is likely not to be; therefore, the spare block can be modified so that the XOR operation between primary and spare bits recovers the original value stored in the cell. In the unlikely case that both cells are stuck, ECP entries in the spare block are used to correct the stuck cell pair.

ZombieECP. This simple extension of ECP [30] works as follows. Once blocks are paired, the spare block stores additional ECP entries to correct stuck bits both in the primary block and in the spare block itself. This mechanism is adaptive, so blocks are divided into subblocks. When the level of failures in a primary block is low, a small spare subblock provides sufficient resources for correct operation. As failures accumulate, larger subblocks are used, up to a full block.

ZombieERC. This mechanism applies erasure codes that take the location of stuck-at bits into account [34]. It uses one-to-many encodings of values and chooses an encoding that matches the values in stuck cells. Providing one-to-many encodings requires additional storage, which is gradually supplied by spare subblocks or blocks as failures accumulate. ZombieERC prefers encodings that least modify primary bits, shifting the wear to spare bits.

ZombieMLC. ZombieMLC uses different coding solutions depending on the number of stuck cells. Under no failures, the simplest codes, which are based on rank modulation and only tolerate drift, use a one-to-one mapping of messages to balanced codewords, i.e., strings where all ranks, or *coordinate values*, occur equally often. As permanent failures accumulate, ZombieMLC switches to one-to-many encodings; these encodings not only take advantage of the likelihood of cells drifting together, like rank modulation does, but they also use predefined anchor values to determine how to unshuffle the codeword to recover the original string.

4. IMPLEMENTATION DETAILS

Zombie memory controllers transparently offer unmodified read and writeback semantics to last level caches. Zombie adds features only to the control path and error correction modules of memory controllers. Some of these features, such as those related to spare pool management, are common to all Zombie mechanisms. Others are specific to particular mechanisms. Note that ZombieMLC uses a different block size than ZombieSLC, but it still stores 512 bits of data.

4.1 Common Zombie Functionality

The Zombie framework provides common block formats, structures and mechanisms.

A. Block Formats

Primary block format. For ZombieSLC, the ECP metadata in the primary block has a field to indicate whether all entries are used, as in the original ECP proposal. Zombie adds a two-bit field to indicate either that the block is unpaired or the size of the subblock in use. The remaining ECP bits are used to redundantly store the spare block address and additional metadata. ZombieMLC does not use ECP, so the pointer is stored in bits previously used for data or in reserved fields.

Spare block format. Spare blocks are one of three sizes, the largest being a full block (512-bits). Intermediate sizes depend on the Zombie mechanism being used. When the block is divided into subblocks, the memory controller records which subblocks have been allocated and which are free, either in the spare block itself or in a separate table.

B. Basic Structures

Spare block pools. The memory controller keeps a separate pool for each subblock size. Pools are simply a head entry pointing to block linked-lists implemented using the storage in the blocks themselves.

Spare data buffer. The memory controller's control path and error correction modules share and communicate spare block contents through this buffer.

C. Basic Mechanisms

Locating spare blocks. When a paired primary block is accessed, the memory controller's control path module locates the spare block by following the pointer stored in the primary block. When a block is read, it places the spare block data into the spare data buffer. When a block is written, it copies the contents of the spare data buffer to the spare block.

Managing free spare blocks. On block failure, the memory controller finds the smallest available subblock of appropriate size and pairs it with the primary block; it then updates the corresponding spare block allocation bit and removes the block from the pool if it has been completely paired. If there are no spare blocks in any pools, the entire page containing the failing block is disabled, and its blocks are divided into the smallest supported subblock size and inserted in the pool. If the primary block had previously been paired with a smaller subblock, this spare subblock is returned to its pool. As larger subblocks are needed, the memory controller combines smaller contiguous subblocks into larger ones.

4.2 ZombieSLC Mechanisms

ZombieSLC mechanisms use existing schemes for intrinsic block error correction. For example, we assume ECP with six entries (ECP6). With ECP6, when a block suffers its seventh bit failure, it is paired, and a ZombieSLC mechanism becomes active for that block.

4.2.1 ZombieXOR Mechanism

ZombieXOR is the simplest Zombie strategy. It pairs two blocks cell-by-cell using XOR operations (see Figure 4). ZombieXOR is not adaptive, so primary and spare blocks are always the same size. The semantics of reading and writing the block changes for paired blocks: A cell of data is the XOR of cells with the same offset in primary and spare blocks. ZombieXOR repurposes ECP entries in the spare blocks to replace cells in offsets where both the primary and spare data cells have failed. It adds a bit-wise XOR unit to the memory controller's error correction module.

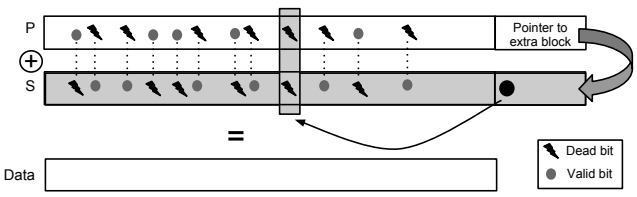


Figure 4: Blocks paired with XOR operations. If aligned bits have both failed, an ECP entry in the spare block replaces those bits.

Read operation. Reads are straightforward: If the block is paired, both the primary and spare blocks are read from memory. The primary block P is read as is, and the spare block S is read and corrected by its ECP entries. The original data is finally obtained by performing a bit-wise XOR of these values. Reading each paired block requires reading

two blocks from main memory. However, reads of primary and spare blocks can proceed in parallel if the *pairing* information (not the data) is cached by the memory controller and the blocks have been mapped to independent banks.

Write operation. Writes are more complex: The memory controller must determine what values need to be written to the primary (P) and spare (S) blocks. The goal is to write two values, P' and S', such that $P' \oplus S'$ encode the desired data A (see Figure 4). To reduce wear, ZombieXOR writes mostly to S and only writes to P bits that, when XORed with corresponding S' stuck-at bits, recover A's bit. This lengthens P's lifetime at the expense of S, but a spare block can be easily replaced, while a primary one cannot. If writes to P also fail, the memory controller allocates an ECP entry in the spare block for each failing bit (rarely more than one at a time) and retries the write. If S's ECP entries are exhausted, the memory controller allocates a new spare block. If the new pairing fails, P's page is disabled, and its blocks are added to the spare pool.

4.2.2 ZombieECP Mechanism

ZombieECP is an adaptive extension of ECP that uses the space in a spare subblock for additional ECP entries. As in the original ECP proposal, ZombieECP entries are used to correct stuck-at data cells or to replace another failed ECP entry. The subblock size gradually increases on demand. For SLC cells, 128-bit subblocks support up to 12 entries, 256-bit subblocks support up to 25, and 512-bit blocks support up to 51.

ECP uses an active entry count to determine which entries are already in use. This count must be larger for ZombieECP. When the primary block gets paired, its entries are copied to the spare block, and its ECP metadata field is repurposed to store the spare subblock pointer and other metadata, such as the combined entry count. Read and write operations are performed as in ECP except they use a larger number of ECP entries. Though its hardware is similar to ECP, ZombieECP includes a log-depth parallel prefix circuit to keep latency low.

4.2.3 ZombieERC Mechanism

ZombieERC divides blocks into multiple data messages of size k and encodes them into larger codewords of size n with an erasure code $[n, k, d]$ that can tolerate up to $d-1$ errors. This code uses the knowledge of stuck-at bit locations, as described by Tsybakov [34]. This type of code provides a one-to-many mapping of data messages into codewords. Having multiple codewords to choose from lets ZombieERC tolerate stuck-at cells by finding a compatible codeword, i.e., a codeword containing symbols that match the stuck-at cell values.

ZombieERC extends Tsybakov's proposal with a hardware-optimized implementation using three optimizations: (1) biasing writes toward the spare block, (2) adapting to a gradually higher number of failures by increasing the subblock size to accommodate more error tolerant encodings, and (3) calculating one-to-many mappings using a table-based implementation.

After encoding, the k first bits of each codeword are stored in the primary subblock, and the remaining in the spare. ZombieERC intentionally partitions codewords into primary and spare blocks, and it biases the encoding so that bit flips are more frequent in the spare block, unlike Tsybakov's coding scheme. This is done to preserve primary blocks: A spare block can be readily replaced with another, while a primary one cannot. ZombieERC is also adaptive; thus, as failures accrue, subblocks grow, stretching to make space for more error tolerant encodings. For SLC PCM, ZombieERC uses 128-bit subblocks to tolerate up to 2 errors per codeword, 256-bit subblocks to tolerate up to 3 errors per codeword, and 512-bit blocks for 3 errors per *smaller* codeword.

Typically, using Tsybakov's codes requires solving a system of linear equations. Instead of implementing hardware to solve this system dynamically, we partition blocks into smaller data messages, which reduces the size of matrices used for decoding and their inverses used for encoding. These smaller matrices can be pre-computed for every set of stuck-at cells and allow for a table-based implementation that removes the need to solve the system dynamically, resulting in a hardware-optimized design. For SLC PCM, ZombieERC encodes 20-bit messages into 25-bit codewords (128-bit subblocks, 2 errors per codeword), 10-bit messages into 15-bit codewords (256-bit subblocks, 3 errors per codeword), and 4-bit messages into 8-bit codewords (512-bit blocks, 3 errors per codeword). The total memory required to store all matrices is less than 16KB.

As noted, ZombieERC relies on a priori knowledge of stuck-at locations. A naïve approach to generating this knowledge is to flip all cells and read them back, comparing new to original values. However, performing this operation would cause too much wear. Instead, ZombieERC employs a failure location cache, similar to previous work [31], to mitigate excessive wear. This cache consists of entries that store a bit vector for error locations in both primary and spare blocks, and it has overheads similar to the cache presented in prior work [31].

4.3 ZombieMLC Mechanism

ZombieMLC, a new encoding mechanism, tolerates *both* stuck-at failures and drift. It combines ideas from rank modulation with a novel form of encoding information about failures based on the position of special symbols, called *anchors*. This encoding assumes that the two possible types of stuck-at cell failures can be differentiated and that they represent the lowest and the highest symbol values.

As discussed in Section 2, rank modulation constructs a codeword as a string of relative ranks of values stored in a group of cells. A contrived example of a 3-cell group illustrates the concept. The first cell has the highest value (rank 5), the second cell has the lowest value (rank 1), and the third cell has the middle value (rank 3), resulting in the string r5,r1,r3 (note that a string is a sequence of ranks). This permutation of ranks, instead of the absolute cell values determined by the cell resistance levels of the group, determines the original data message.

ZombieMLC tolerates stuck-at failures by picking anchor(s) with unique ranks and known starting positions embedded within the string. For $d-1$ stuck-at cells, we pick $d-1$ symbols used as anchors that are neither the lowest nor the highest symbol values. The rank-modulated string is appended to the anchors and further shuffled to align the lowest and highest ranks with failures. The known anchor values and respective positions are then used to unshuffle the stored value and recover the original string. For example, if the goal is to correct two errors in the preceding rank-modulated string, ZombieMLC adds two anchors (and two

cells) to the group, as the second and fourth highest values, and prepends them to the string: r2,r4 — r5,r1,r3. Assume there is a 'stuck-at one' failure in position 4 and a 'stuck-at zero' failure in position 5. By shifting the string one position to the right, ZombieMLC obtains a permutation that covers the error locations with the appropriate values (position 4 with the highest and position 5 with the lowest): r3,r2,r4,r5,r1, which is finally stored in memory. At decode time, the position of the first anchor value determines how much to shift the string to recover the original string.

In reality, ZombieMLC uses slightly more complex functions to shuffle the coordinates of the string. It always uses the same number of anchors as there are error locations to correct and covers the error locations with their known values. The problem construction (i.e., codeword length restrictions, shuffle equations, and number of anchors) guarantees unique solutions for every encoding, making it possible to decode what is written to memory.

If each symbol can occur only once in a group, the overhead of permutations could become very large. We use strings to reduce it, allowing multiple symbols to appear uniformly (2-bit MLC) or non-uniformly (3-bit or greater MLC) in the group of cells. For example, for 2-bit cells, groups of 8 (e.g., [0,0,1,1,2,2,3,3]), 12, 16, or more cells trade off *encoding overhead* — since longer codewords reduce the encoding overhead, or *stretch* — and *encoding complexity*, or the computational effort required to encode or decode the codeword. Table 1 lists candidate codes and points out the baseline encoding (B, drift-tolerant, no error correction) and adaptive coding sequences we evaluate (1, 2, 3).

Notation	Codeword	Msg	Errors	Stretch
2-bit MLC (MLC2)				
$[8,5,1]$	8	5	0	1.60
$[12,9,1]$	12	9	0	1.33
$[16,12,1]$	16	12	0	1.33
$[20,16,1]$ (*B,1*)	20	16	0	1.25
$[8,4,2]$ (*3*)	8	4	1	2.00
$[12,8,2]$ (*2*)	12	8	1	1.50
$[16,11,2]$	16	11	1	1.45
$[20,15,2]$	20	15	1	1.33
4-bit MLC (MLC4)				
$[32,25,1]$	32	25	0	1.28
$[48,40,1]$	48	40	0	1.20
$[64,55,1]$ (*B*)	64	55	0	1.16
$[32,24,2]$	32	24	1	1.33
$[48,39,2]$ (*1*)	48	39	1	1.23
$[64,52,2]$	64	52	1	1.23
$[29,20,3]$	29	20	2	1.45
$[41,30,3]$ (*2*)	41	30	2	1.37
$[57,45,3]$	57	45	2	1.27
$[28,17,4]$ (*3*)	28	17	3	1.65
$[42,30,4]$	42	30	3	1.40
$[54,40,4]$	54	40	3	1.35

Table 1: String stretch for 2-bit and 4-bit MLC PCM. (B=baseline code, and the numbers show the codes used for our adaptive encoding scheme.)

Two main steps are required for encoding ZombieMLC codewords: rank modulation and stuck-at cell error correction. First, ZombieMLC converts the message into a drift-tolerant codeword (string). Second, it layers error correction on top of the string by prepending more (unique) symbols, the anchors, to the string and further shuffling the codeword to tolerate stuck-at cell failures if the region of memory to be written contains them. Knowing the anchor values and locations provides an 'undo' function that is used to decode the codeword. To correct single-cell failures, the anchor value changes, but its string location or position stays the same. To correct two or more cell failures, the anchor values stay the same, but their locations in the string are shuffled so that other known values (not the anchors) can be "written" to the error locations. Figure 5 illustrates these two main steps.

4.3.1 Rank Modulation Step

The goal of step 1, rank modulation, is to use m coordinate values (or ranks, or resistance levels available in a cell) and a group of $2m$ cells to encode a data message so that the sequence in which these values appear in the group can be used to decode the message[3]. Each cell is permanently assigned a coordinate $1 < c < 2m$, which indicates its relative position in its group. Coordinates are shuffled to generate a string. Figure 5 shows an example shuffle of the data message, the decimal number '1,001', into the rank-modulated string [3,0,1,2,2,1,0,3], using well-known techniques. We refer the reader to Barg et al.'s work [4] for details on how the encoding and decoding of rank modulated-strings is done.

4.3.2 Stuck Cell Error Correction Step

The second step depends on how many stuck-at cells need to be accommodated. Items 2a and 2b in Figure 5 show the process for one and two stuck-at cells, respectively.

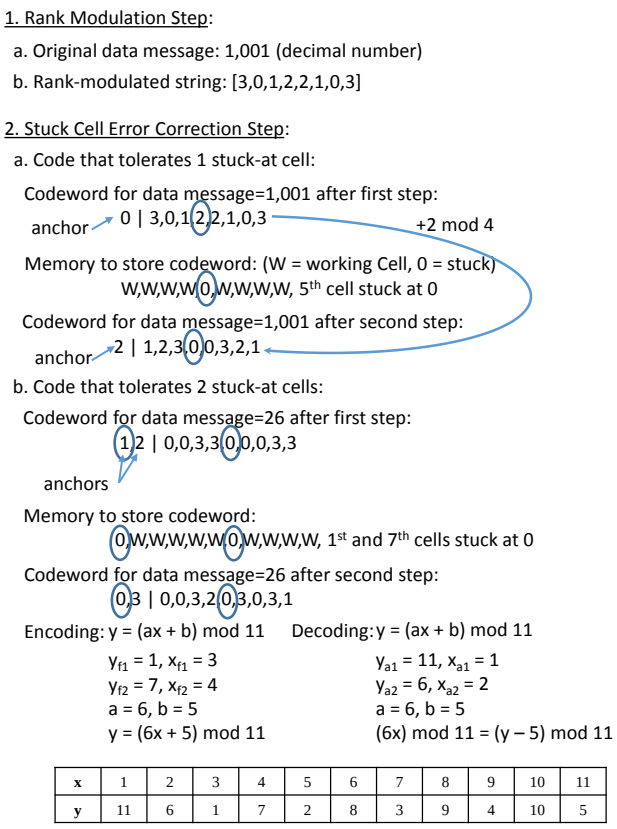
For one stuck-at cell, say code [9,5,2] for 2-bit MLC, ZombieMLC prepends a single symbol of known coordinate value (e.g., 0) to the string that represents the data value after rank modulation. It then subtracts the original value to be stored in the stuck-at cell (2, circled in 2a) from the faulty cell value (0 in this example). This value is added to each symbol in the string. Both subtraction and addition are modulo the number of levels these cells support. This finally produces the codeword to be written in memory (2a, bottom).

For two and three stuck-at cells, ZombieMLC prepends as many anchors as stuck-at cells to the rank-modulated string, resulting in a new string s. The rank-modulated string contains non-anchor symbols, where each symbol occurs a (nearly) equal number of times, and both lowest and highest possible cell values occur at least as many times as stuck-at cells to be tolerated; otherwise, there may not be enough of these values to map to stuck-at cells.

If ZombieMLC were to write s unmodified to memory, the presence of stuck-at cells could cause the arbitrary values in s to be corrupted because one or more of its symbols cannot be written. Thus, ZombieMLC shuffles this string using a permutation on its indices (i.e., symbols in s are placed in new locations within that group of cells) so that the stuck-at cells are aligned with the symbols these cells return when read. For example, if a location is stuck at 0, the value mapped to that index is also 0. The mapping function ZombieMLC uses to generate this permutation belongs to a carefully crafted family, which guarantees that the permutation always exists and is easy to compute. To decode, ZombieMLC inverts this permutation. The family of mapping functions we use allows ZombieMLC to use the knowledge of anchor locations (due to their unique values)

[3]Other ratios of ranks to cell group size are possible.

in the string read from memory to invert the permutation. For two errors, n (the size of the string to be written in memory) must be a prime power and the family of mapping functions consists of linear functions $y = ax + b$, where the arithmetic is over a finite field of size n, x is a location in a rank-modulated string, and y is the location where it is stored in memory. For three errors, n is a prime power plus one, F is a finite field of size $n - 1$, and the family of functions consists of Mobius transformations of the form $y = (ax + b)/(cx + d)$ with $ad - bc = 1$, over the projective line $F \cup \{\infty\}$. Alon and Lovett [2] provide a more detailed discussion.



x	1	2	3	4	5	6	7	8	9	10	11
y	11	6	1	7	2	8	3	9	4	10	5

Figure 5: Two-step ZombieMLC string encoding and decoding process: rank modulation and correction for one and two stuck-at cells. 'W' denotes working cells.

Item 2b in Figure 5 shows the encoding of a codeword that tolerates up to two stuck-at cells. Anchors with values 1 and 2 are prepended to the rank-modulated string. The first and seventh positions in the memory that will be used to store the final codeword are stuck at 0 (circled in the figure), so $y_{f1} = 1$ and $y_{f2} = 7$. First, ZombieMLC finds the two leftmost positions with values at the same level as the stuck-at cells, the third and fourth positions in this example. These are the positions the mapping function should shuffle into the stuck-at cells. Thus, $x_{f1} = 3$ and $x_{f2} = 4$. Next, ZombieMLC computes a and b such that $y_{f1} = (ax_{f1} + b)$ mod 11 and $y_{f2} = (ax_{f2}+b)$ mod 11, which results in $a = 6$ and $b = 5$. Note that because n, the codeword length, is prime, modulo arithmetic and finite field arithmetic are the same. We refer readers to Horowitz's work [11] for details on finite field arithmetic. Finally, a and b are used to determine the final coordinate for every symbol in the string (Figure 5, mapping table at the bottom), and values are written to memory according to this shuffle (see codeword after second step in Figure 5, 2b). To decode, ZombieMLC uses the knowledge of the anchor values (1 and 2), their original locations at the beginning of the string ($x_{a1} = 1$ and $x_{a2} = 2$), and their current locations ($y_{a1} = 11$ and $y_{a2} = 6$) to recover a and b and revert the encoding shuffle. Once unshuffled, the string goes through rank demodulation to recover the data message (26). A more complete set of encoding and decoding examples, including rank modulation is provided elsewhere [8].

4.4 Zombie Hardware and Memory Operations

Zombie requires only minor hardware changes, summarized below, along with additional memory operations that affect Zombie's performance only when blocks are paired.

Memory controller. The logic and memory required in the memory controller are small and add only one cycle to the control path. ZombieERC's erasure code table is read-only and at most 15.6 KB for the codes we use, smaller than a typical L1 cache. The largest ZombieERC codeword size we use is 25 bits, requiring five levels of simple logic for the bitwise multiplication (an XOR). For ZombieMLC, we use multiple simple functional units to solve the sequence of equations required to generate the string. A small collection of multiplexers with at most 6 levels is required to shuffle cells to transform the string into the codeword and vice versa.

Memory operations. Operations over paired blocks require at least one additional memory operation. On reads, the memory controller must read both primary and spare blocks to reconstruct the original data. Writes are more complicated: For mechanisms that use the error location cache, a miss in this structure requires flipping all bits in the block to recover the location of failed bits. All mechanisms require writes to both primary and spare blocks. If new bits fail during these writes, additional writes are required. Table 2 summarizes the number of additional operations required in the worst-case.

Mechanism	Additional	
	Read Ops	Write Ops
ZombieECP	+1 RD(S)	+1WR(S)
ZombieERC	+1 RD(S)	+1WR(P),+2WR(S)
ZombieXOR	+1 RD(S)	+1 RD(S), +1 WR(P)
ZombieMLC	+1 RD(S)	+1 WR(S), +1 WR(P)

Table 2: Additional operations required for the various Zombie mechanisms. (S:Spare and P:Primary blocks)

5. EVALUATION

We use two kinds of simulation approaches to model Zombie: (1) a *cache and memory simulator* to collect information about cache and bit/cell flipping behavior, and a *statistical simulator* to estimate lifetime. The cache and memory simulator is based on Pin [21] and simulates a cache hierarchy of 64KB, 8-way set associative data L1 cache, a shared 2MB, 8-way set associative L2 cache, and main memory. We run SPEC2006 workloads [10]. On an average write, the raw

data bit flip rate is only 17% of bits in a block. For SLC, this naturally translates into a 17% cell flip rate. As bits are grouped into multi-bit cells (MLC), the cell flip rate increases because cells have to be re-written every time any of their bits flip. We have measured an average of 31% and 39% cell flip rates for 2-bit and 4-bit cells, respectively. Additionally, once raw data gets encoded, flip rates grow significantly. We use 75% and 93.75% flip rates to account for the wear of rank modulation encoding of 2-bit and 4-bit cells for our baseline and error-corrected data, respectively.

The statistical simulator uses a methodology very similar to ECP [30]: it simulates at least 10,000 pages, assuming a normal bit failure distribution with a mean of 10^8 write operations, 0.25 coefficient of variance, and wear leveling across pages in memory, blocks in pages, and bits in blocks. ZombieSLC mechanisms rely on ECP to tolerate the first 6 failures in a block. Average results are reported without variance, which is always under 3%. Endurance improvement is the ratio of aggregate writes performed by a mechanism by the time memory drops below 24% capacity.

Table 3 lists the various SLC failure tolerance mechanisms against which we compare Zombie mechanisms.

Name	Approach
NoCorrection	Discards block on first error
SEC	Single Error Correction per 64-bit (ECC)
DRM [15]	Pairs pages, corrects up to 160 errors per *page*
ECP [30]	Error Correcting Pointers. Default: corrects up to 6 errors per block
FREE-p [37]	Corrects up to 4 hard and 2 soft errors per block and remaps blocks
SAFER [31]	Corrects up to 32 errors per block
PAYG [24]	Hierarchical ECP
Oracle64	Magically tolerates 64 errors per block
Oracle128	Magically tolerates 128 errors per block
ZombieECP	Increasing number of ECP entries over time; pairing with 128-bit, 256-bit and 512-bit blocks
ZombieERC	Increasing erasure code strength over time; pairing with 128-bit, 256-bit and 512-bit blocks
ZombieXOR	Pairs 512-bit blocks, up to n tries on pairing
ZombieMLC	Increasing code strength entries over time; pairing with 64-bit, 128-bit, and 512-bit blocks

Table 3: SLC and MLC PCM lifetime extension mechanisms evaluated.

5.1 Overall Zombie Lifetime Improvements

ZombieMLC. Figure 6 shows results for ZombieMLC. Baselines (MLC-2 and MLC-4) perform rank modulation (i.e., are drift-tolerant), but they cannot tolerate stuck-at failures so their memory capacity degrades very quickly[4]. In contrast, ZombieMLC ($MLC2_{Adapt}$ and $MLC4_{Adapt}$) tolerates both drift and stuck-at failures and therefore significantly extends the lifetime of MLC PCM (17× and 11×, respectively).

ZombieSLC. Figure 7 shows how memory capacity degrades as pages are written and bits fail due to wear for ZombieSLC. Overall, ZombieSLC mechanisms provide high memory capacity longer than any other previously proposed

[4]MLC-2 degrades more quickly than MLC-4 because it uses a larger number of cells per block; therefore, the probability of having a failure in at least one cell is higher.

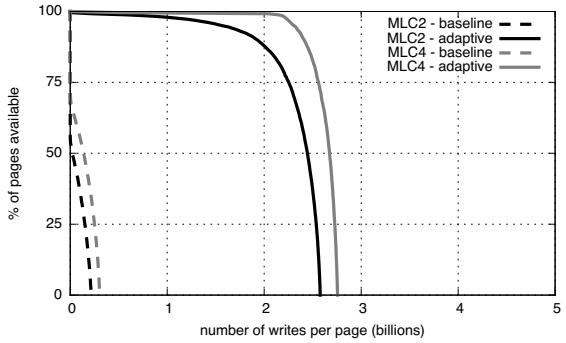


Figure 6: Memory capacity degradation due to failures that cannot be tolerated as memory is written over its lifetime for 2-bit and 4-bit MLC PCM.

lifetime extension mechanism. This results from reviving dead blocks to extend the lifetime of live blocks.

In general, ZombieSLC has a four-phase life cycle. The first phase corresponds to the portion of lifetime covered by ECP or some other intrinsic error correction scheme. Once bit failures exceed what ECP can tolerate, the second phase begins: Failed blocks now start being paired. By enabling subblock pairing, we can maintain higher available memory capacity longer, handling more errors gracefully. The third phase is defined by all blocks being paired. The fourth phase begins when paired blocks can no longer correct the errors: Spare blocks are retired (really dead) and primary blocks are added to the pool (not dead yet). Recall that all ZombieSLC mechanisms bias bit flips to the spare blocks, so primary blocks may still have a useful Zombie lifespan even after their spare blocks have died.

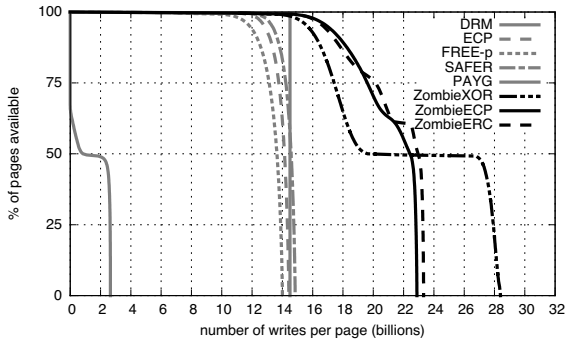


Figure 7: Memory capacity degradation due to failures that cannot be tolerated as memory is written over its lifetime, for multiple failure tolerance mechanisms: DRM, ECP, FREE-p, SAFER, PAYG, ZombieECP, ZombieERC, and ZombieXOR.

The first knee of the curve in Figure 7, at 98% memory capacity, demonstrates how little memory capacity must change before ZombieSLC shows an improvement over the intrinsic error correction capability of PCM, like ECP or SAFER. Further, in phase two, ZombieSLC's rate of memory capacity degradation is lower than other mechanisms. Specifically, PAYG fails on the first uncorrectable cell (i.e.,

from 100% to 0% instantly); ECP, DRM and SAFER fall from 100% to 50% capacity in about 1 billion writes, and FREE-p in about 2.3 billion writes. This transition happens over 3.8 billion writes for ZombieXOR, 10 billion writes for ZombieECP, and 11 billion writes for ZombieERC. ZombieXOR maintains 50% memory capacity longer than any other mechanism. Finally, ZombieXOR's rate of memory capacity degradation, from 50% to 0% capacity, is lower than any other mechanism's; in all of PCM's operational regions, ZombieSLC is superior to all other mechanisms.

ZombieXOR. This all-or-nothing mechanism trades simplicity for pairing granularity (only full blocks are paired). It achieves a 92% endurance improvement over PAYG, the best prior mechanism. ZombieXOR provides little memory capacity boost in phase one. Blocks gradually get paired in phase two as the number of failures in a block exceeds ECP's correction capacity (6 failures). By the end of this phase, ECP, PAYG, FREE-p and SAFER have already completely exhausted their memory capacity. In phase three, all ZombieXOR blocks are paired, so memory remains at 50% capacity for a long time. The reason is the redundancy provided by paired bits and the additional resiliency of revived ECP entries in the spare block. Once enough pairs of bits fail, spares are discarded, and memory capacity starts dropping again (fourth phase).

ZombieECP. Due to variation in the endurance of individual cells, outlier failures prematurely reduce memory capacity. ZombieECP gradually extends ECP by using variable-size subblocks. Spare subblocks of three possible sizes can simultaneously co-exist in the system, so there are no well-defined stable phases like ZombieXOR's. Overall, ZombieECP achieves a 58% endurance improvement over PAYG.

ZombieERC. Like ZombieECP, ZombieERC gradually introduces increasingly greater error tolerance by growing its subblock size, so phases are again not well defined. It achieves 62% longer endurance than PAYG. At 17% raw data bit flip rates, ZombieERC is approximately equivalent to ZombieECP. This is because the coding itself causes additional flips and raises ZombieERC's effective wear rates (after encoding) to higher levels.

ZombieSLC sensitivity to cell flip rate. PCM endurance is sensitive to cell flip rates. In general, curves are scaled to the left and down (lower endurance) with increased cell wear rates. One notable exception is ZombieERC, which improves relative to other ZombieSLC mechanisms as the wear rate increases. At the measured 17%, ZombieERC's adds bit flips due to its encoding. As raw data flip rates increase, ZombieERC's overall wear is not affected as much as other ZombieSLC mechanisms, again due to its encoding. As a result, if memory were to be compressed or encrypted (raw data bit flips around 50%), ZombieERC would be the most effective correction mechanism.

Zombie error location cache sensitivity. We conducted a cache sensitivity analysis by varying the error location cache size from 32 K entries up to 256 K entries (powers of two) and its associativity from 4- to 8-way. Most SPEC2006 benchmarks have error location cache miss rates below 1% with a 8-way 256K error location cache, never exceeding 2%.

5.2 A Closer Look at Zombie Lifetimes

Table 4 shows the normalized average number of writes per page until memory capacity falls to the 98%, 49%, 24% and 0% thresholds. SLC lifetimes are normalized to SEC to demonstrate the relative improvements of the various mechanisms. ZombieMLC lifetimes are normalized to respective baselines, which provide no stuck-at cell correction, only drift tolerance.

Mechanism	Writes per page				% Cell failures
	98%	49%	24%	0%	
NoCorrection	-	-	0.0	0.1	0.0%
SEC	1.0	1.0	1.0	1.0	0.1%
DRM	-	0.3	0.4	0.4	0.5%
ECP	5.2	2.5	2.4	2.3	0.5%
FREE-p	4.7	2.4	2.3	2.3	0.7%
SAFER	5.4	2.5	2.5	2.4	0.9%
PAYG	6.1	2.5	2.4	2.4	0.4%
Oracle64	10.3	4.4	4.2	4.1	9.5%
Oracle128	12.4	5.2	5.0	4.9	20.8%
ZombieECP	6.7	3.9	3.8	3.7	5.1%
ZombieERC	6.7	4.0	3.9	3.8	13.8%
ZombieXOR	6.2	4.7	4.6	4.6	19.6%
ZombieSAFER	6.4	4.1	4.2	4.1	15.2%
ZombieMLC2$_{adap}$	-	89.6	16.6	12.0	0.5%
ZombieMLC4$_{adap}$	-	18.7	10.9	9.1	0.5%

Table 4: Average number of writes per page (normalized with respect to SEC for SLC and drift tolerance for MLC) until memory is reduced to 98%, 49%, 24%, and 0% of its original capacity, and the total fraction of cells that failed during the memory lifetime.

These results show that PAYG achieves a lower number of total cell failures than ECP, in contrast with data provided in the original PAYG proposal. The reason is a methodological difference. The original PAYG proposal assumes a system stops operation at the first uncorrectable failure for both PAYG and ECP. Although PAYG can no longer correct additional errors at this point, ECP can still degrade memory capacity after the first uncorrectable failure because there are many other blocks with unused error correction entries. Thus, we continue simulation of ECP beyond this first failure, which provides additional wear and resulting opportunities for new failures to appear.

For all capacity levels in Table 4, Zombie results in the largest number of writes among all realistic failure tolerance mechanisms. However, the best ZombieSLC mechanism is not the same for all phases. Therefore, the best mechanism must be selected depending on design goals. If the goal is to keep memory close to 100% capacity for the longest time, a designer should select ZombieERC or ZombieECP. On the other hand, if the goal is to keep memory capacity above 50% for the longest time, ZombieXOR is a better option. Finally, the increased density and the need for drift tolerance of MLC PCM dramatically reduces lifetime compared to SLC, but ZombieMLC reverts this reduction in lifetimes by over an order of magnitude.

5.3 Zombie Performance

Zombie's performance degradation is very low, even after the previous proposals have exhausted their lifetimes. Figure 8 shows the execution time of SPEC2006 workloads for ZombieSLC normalized to a system with no pairing. By the time the best alternative method — PAYG (vertical line in Figure 8) — fails, the slowdown of ZombieSLC is a negligible 0.1%, on average. The reason for such low overhead is the negligible latency the extra control path logic adds to operations on unpaired blocks.

By the time *all* blocks are paired (worst possible case), the slowdown is significantly higher due to the additional memory operations to spare blocks, but it is still not prohibitive, at 6-10%. Likewise, Figure 9 shows that ZombieMLC also incurs very little performance degradation by the time its counterpart fails, as well as tolerable performance degradation at the end of its own lifetime.

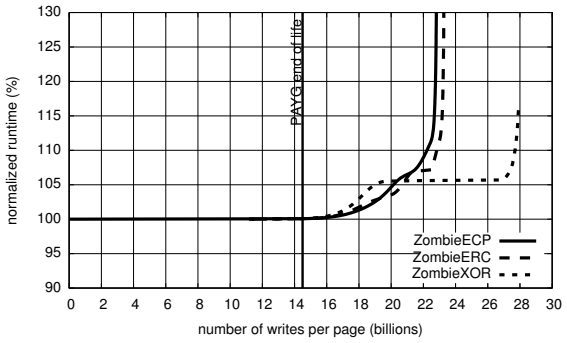


Figure 8: ZombieSLC performance over time as the memory system degrades. The vertical line shows where the best alternative life extension technique fails.

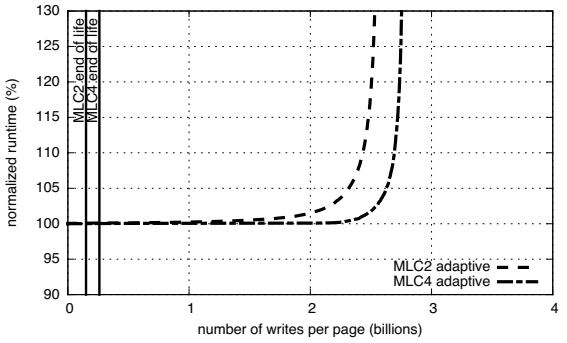


Figure 9: ZombieMLC performance over time as the memory system degrades. The vertical lines show where the baseline rank modulation fails.

6. RELATED WORK

Three overarching strategies mitigate the problem of limited cell lifetime. The first reduces the total number of PCM writes. A system can avoid bit-cell writes using a variety of techniques: adding buffers in front of PCM to absorb temporally local writes [26, 20, 39], writing only cells whose bits have actually changed [35, 40, 20, 38], and dynamically adapting the data encoding used based on past behavior [6, 38, 27, 17]. Given that writes are eventually necessary, the second strategy attempts to spread them out evenly across the entire memory system. These wear-leveling techniques [40, 39, 25, 32] can prevent hot spots of activity from occurring, thus preventing excessive wear and early exhaustion of particular bits, blocks and pages. Finally, given that failures will eventually occur, the last strategy is to add some amount of redundant cells or redundant information so that failures can be detected and corrected [15, 30, 31, 37, 24, 7, 17]. Although Zombie belongs to the third category, many of these solutions can be easily combined with the Zombie framework.

Theoretical investigation of coding for memory with stuck-at locations was initiated by Kuznetsov and Tsybakov [19, 34], with some recent progress [9]. However, we found the early papers [19, 34] to be more applicable to our setting of short message and codeword lengths. Rank modulation in the presence of errors has been addressed by Barg and Mazumdar [4], and Yehezkeally and Schwartz [36], but they do not use a stuck-at fault error model.

DRM [15] pairs two pages with failures to make one working page. Like Zombie, DRM pairs memory regions, although DRM does it at coarser granularity (pages vs. blocks). SAFER [31] takes advantage of PCM's stuck-at bit failures by partitioning a data block dynamically; each partition has at most one failed bit. An additional bit indicates whether the bits in a partition have to be inverted when they are read. ECP [30] is an error correction mechanism that corrects cell failures by using pointers to failed bits and replacement bits to correct them. ZombieSLC may use SAFER or ECP as a block-intrinsic error correction mechanism, among others. ZombieECP also extends ECP with additional entries. Like Zombie, FREE-p [37] pairs failed blocks to working blocks. However, FREE-p uses failed block cells to store only a pointer to the working block, while Zombie uses them for data and error correction, which is a more efficient use of cells. PAYG [24] proposes that all blocks have only one dedicated ECP entry and that other entries be part of a common pool. Once all entries in this pool are exhausted, the system fails without attempting to reuse partially worn blocks of memory to gradually degrade capacity, unlike Zombie. Further, instead of obtaining entries from a shared pool of pristine entries, like PAYG does, ZombieECP obtains them from a failed page. AECC [7] is an adaptive mechanism that changes the allocation of bits within a block to store data and metadata. Zombie adaptively increases metadata storage with external blocks and subblocks. FlipMin [17] is a concurrently proposed mechanism that, like ZombieERC, is based on the theory put forward by Kuznetsov and Tsybakov [19, 34]. Both select nearest codeword matches in the presence of stuck-at faults and bias writes away from bit flips. Unlike FlipMin, ZombieERC pairs blocks, directs wear to the spare block, and is adaptive.

Even though the preceding mechanisms have improved PCM lifetimes, pages, often disabled with many working cells left, are simply wasted. Zombie dramatically increases bit lifetimes by reusing working bits in the blocks on pages that have been disabled due to failures in only some of their bits.

Table 5 compares these algorithms along multiple dimensions. The *storage* column refers to the location and granularity of metadata: a separate page, a separate block, a separate subblock, or additional bits in the original block (intra-block). The *method* refers to how error tolerance is achieved: (summ) by summarizing (*e.g.*, erasure codes, ECC); (pair) by pairing regions of memory (*e.g.*, XOR, DRM, FREE-p); (repl) by replacing regions of memory that no longer work with another (*e.g.,* ECP, FREE-p); or (part) by using partitioning with (inv) bit inversion per partition (SAFER). The third column (*Adpt*) refers to whether the mechanism can grow its error tolerance dynamically by distributing data

Mechanism	Storage	Method	Adpt	OS	Rus
SEC	intrablock	summ	no	no	no
DRM [15]	page	pair	no	yes	yes
ECP [30]	intrablock	repl	no	no	no
SAFER [31]	intrablock	part/inv	no	no	no
FREE-p [37]	block	pair/repl	no	yes	no
PAYG [24]	subblock	repl	yes	no	no
AECC [7]	intrablock	summ	yes	yes	no
FlipMin [17]	intrablock	summ	no	no	no
ZombieECP	subblock	pair/repl	yes	no	yes
ZombieERC	subblock	pair/summ	yes	no	yes
ZombieXOR	block	pair	no	no	yes
ZombieMLC	subblock	pair/summ	yes	no	yes

Table 5: Comparison to related work.

and metadata differently. The *OS* category refers to whether OS support is required beyond what is already provided in current DRAM systems. DRM relies on the OS to pair pages based on error locations; FREE-p relies on the OS to manage spare blocks; and AECC relies on the OS to switch to stronger codes by remapping data and metadata bits. The *Rus* column refers to the reuse of pages and blocks that were disabled due to failures that could not be corrected. As noted, the only mechanisms that support this feature at fine granularities are those proposed in this paper.

7. CONCLUSIONS

This paper proposed Zombie, a framework that can be used with prior and new error correction mechanisms to significantly improve SLC and MLC PCM lifetimes. Zombie uses memory that has been disabled due to exhaustion of intrinsic block error correction resources to keep memory that is still in service alive longer. Three of these mechanisms — ZombieXOR, ZombieECP, and ZombieERC— show endurance superior to various state-of-the-art SLC PCM error correction mechanisms. These mechanisms can also be used to correct stuck-at failures in MLC PCM, but doing so would require an additional compatible mechanism to tolerate drift. The fourth mechanism, ZombieMLC, is to our knowledge the first proposal to tolerate both stuck-at failures and drift in an integrated and seamless manner. ZombieMLC increases the lifetime of MLC PCM by over an order of magnitude compared to a standard rank-modulation mechanism, which tolerates only drift.

In summary, the Zombie framework enriches the toolbox of designers seeking error correction mechanisms that match their specific system design goals.

8. ACKNOWLEDGMENTS

The authors thank Rodrigo Gonzalez-Alberquilla and Luis Ceze for their contributions to the ideas and initial thinking that led us to this work. The authors also thank the anonymous reviewers for insightful comments. Finally, the authors thank Sandy Kaplan for her effort editing the manuscript. Rodolfo Azevedo is funded by CNPq and FAEPEX.

9. REFERENCES

[1] S. Ahn *et al.*, "Highly manufacturable high density phase change memory of 64mb and beyond," in *Electron Devices Meeting, 2004. IEDM Technical Digest. IEEE International*, Dec. 2004, pp. 907 – 910.

[2] N. Alon and S. Lovett, "Almost k-wise vs. k-wise independent permutations and uniformity for general group actions," in *International Workshop on Randomization and Computation (RANDOM)*, 2012.

[3] G. Atwood, "The evolution of phase change memory," Micron, Tech. Rep., 2010.

[4] A. Barg and A. Mazumdar, "Codes in permutations and error correction for rank modulation," *IEEE Transactions on Information Theory*, vol. 56, no. 7, pp. 3158 – 3165, July 2010.

[5] G. W. Burr *et al.*, "Phase change memory technology," *Journal of Vacuum Science and Technology B*, vol. 28, no. 2, pp. 223–262, 2010.

[6] S. Cho and H. Lee, "Flip-n-write: a simple deterministic technique to improve pram write performance, energy and endurance," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2009.

[7] R. Datta and N. A. Touba, "Designing a fast and adaptive error correction scheme for increasing the lifetime of phase change memories," in *VLSI Test Symposium*, 2011.

[8] J. D. Davis *et al.*, "Supplement to Zombie Memory: Extending memory lifetime by reviving dead blocks," *Technical Report: MSR-TR-2013-47, Microsoft Research Silicon Valley*, 2013.

[9] A. Gabizon and R. Shaltiel, "Invertible zero-error dispersers and defective memory with stuck-at errors," in *International Workshop on Randomization and Computation (RANDOM)*, 2012.

[10] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM Computer Architecture News*, vol. 34, no. 4, Sep. 2006, http://www.spec.org/cpu2006/publications/CPU2006benchmarks.pdf.

[11] E. Horowitz, "Modular arithmetic and finite field theory: A tutorial," in *Proceedings of the second ACM Symposium on Symbolic and Algebraic Manipulation*, ser. SYMSAC '71. New York, NY, USA: ACM, 1971, pp. 188–194. [Online]. Available: http://doi.acm.org/10.1145/800204.806287

[12] Y. Hwang *et al.*, "Full integration and reliability evaluation of phase-change RAM based on 0.24um-cmos technologies," in *2003 Symposium on VLSI Technology*, Jun. 2003.

[13] D. Ielmini *et al.*, "Physical interpretation, modeling and impact on phase change memory (PCM) reliability of resistance drift due to chalcogenide structural relaxation," in *Electron Devices Meeting, 2007. IEDM 2007. IEEE International*, dec. 2007, pp. 939 –942.

[14] ——, "Recovery and drift dynamics of resistance and threshold voltages in phase-change memories," *Electron Devices, IEEE Transactions on*, vol. 54, no. 2, pp. 308 –315, feb. 2007.

[15] E. Ipek *et al.*, "Dynamically replicated memory: building reliable systems from nanoscale resistive memories," in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2010.

[16] ITRS, "Emerging research devices," International Technology Roadmap for Semiconductors, Tech. Rep.,

2009.

[17] A. N. Jacobvitz *et al.*, "Coset coding to improve the lifetime of memory," in *IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[18] A. Jiang *et al.*, "Rank modulation for flash memories," *Information Theory, IEEE Transactions on*, vol. 55, no. 6, 2009.

[19] A. V. Kuznetsov and B. S. Tsybakov, "Coding in a memory with defective cells," *Problems of Information Transmission*, vol. 10, no. 2, pp. 132–138, 1974.

[20] B. C. Lee *et al.*, "Architecting phase change memory as a scalable dram alternative," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, Jun. 2009.

[21] C.-K. Luk *et al.*, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun. 2005.

[22] F. J. MacWilliams and N. J. A. Sloane, *The Theory of Error Correcting Codes*. Amsterdam, New York: North Holland, 1977.

[23] N. Papandreou *et al.*, "Drift-tolerant multilevel phase-change memory," in *Proceedings of the 3rd IEEE International Memory Workshop*, May 2011, pp. 1 – 4.

[24] M. K. Qureshi, "Pay-as-You-Go: Low overhead hard-error correction for phase change memories," in *Proceedings of the 44th International Symposium on Microarchitecture*, 2011.

[25] M. K. Qureshi *et al.*, "Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2009.

[26] ——, "Scalable high performance main memory system using phase-change memory technology," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, Jun. 2009.

[27] ——, "Morphable memory system: a robust architecture for exploiting multi-level phase change memories," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, Jun. 2010.

[28] D. Ralph and M. Stiles, "Spin transfer torques," *Journal of Magnetism and Magnetic Materials*, vol. 320, no. 7, pp. 1190 – 1216, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0304885307010116.

[29] S. Raoux *et al.*, "Phase-change random access memory: a scalable technology," *IBM Journal of Research and Development*, vol. 52, pp. 465–479, Jul. 2008.

[30] S. Schechter *et al.*, "Use ecp, not ecc, for hard failures in resistive memories," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, Jun. 2010.

[31] N. H. Seong *et al.*, "SAFER: Stuck-at-fault error recovery for memories," in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2010.

[32] ——, "Security refresh: prevent malicious wear-out and increase durability for phase-change memory with dynamically randomized address mapping," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, Jun. 2010.

[33] D. B. Strukov *et al.*, "The missing memristor found," *Nature*, vol. 453, pp. 80–83, 2008.

[34] B. S. Tsybakov, "Additive group codes for defect correction," *Problems of Information Transmission*, vol. 11, no. 1, pp. 88–90, 1975.

[35] B.-D. Yang *et al.*, "A low power phase-change random access memory using a data-comparison write scheme," in *IEEE International Symposium on Circuits and Systems*, May 2007.

[36] Y. Yehezkeally and M. Schwartz, "Snake-in-the-box codes for rank modulation," *Information Theory, IEEE Transactions on*, vol. 58, no. 8, Aug 2012.

[37] D. H. Yoon *et al.*, "FREE-p: Protecting non-volatile memory against both hard and soft failures," in *Proceedings of the 17th Symposium on High Performance Computer Architecture*, 2011.

[38] W. Zhang and T. Li, "Characterizing and mitigating the impact of process variations on phase change based memory systems," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2009.

[39] ——, "Exploring phase change memory and 3d die-stacking for power/thermal friendly, fast and durable memory architectures," in *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*, Sep. 2009.

[40] P. Zhou *et al.*, "A durable and energy efficient main memory using phase change memory technology," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, Jun. 2009.